Zircuit OP Bridge

Smart Contract Security Assessment

July 24, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the Zircuit OP Bridge. The protocol modified Optimism's bridge contracts to introduce new mechanisms that give more control over how much funds go through the networks. No major issues were identified during the audit, but some considerations on the protocol level and design were found which mainly concern inconsistencies between the two directions of the bridge and how the new mechanisms are used. The team commented that all the items included in the report do not have a real impact on the protocol and that they can be handled operationally.

BACKGROUND

Zircuit is a fully EVM-compatible zero-knowledge rollup. The code in scope is based on Optimism's codebase. The changes concern the functionality of the bridges which have been modified to support a new mechanism of throttling transactions that go through the bridge. One of the purposes of the throttling is to prevent funds which may originate from hacks go through since in such cases large amounts are expected to be transferred that will be caught by design. On top of this modification, support for finality based on ZK proof verification was also added to the system.

SETTING & CAVEATS

This audit report mainly covers the delta of the changes between a subset of the contracts of the at-the-time private repository zircuit-labs/zkr-op-bridge-audit of the Zircuit OP Bridge at commit `97c73bd621a19393f841c608b1d4b715a3f2f986` and Optimism's official Bridge contracts at Release op-stack v1.7.4 · ethereum-optimism/optimism.

Two auditors worked on the codebase for 2 days on the following contracts:

```
contracts-bedrock/src/
├── L1/
│   ├── L1StandardBridge.sol
│   ├── L2OutputOracle.sol
│   └── OptimismPortal.sol
├── L2/
│   ├── L2StandardBridge.sol
│   └── L2ToL1MessagePasser.sol
└── universal/
    └── StandardBridge.sol
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	Missing finalization throttling on ETH withdrawals	**INFO**
The protocol defines throttles for most of the bridge initiations and the ERC20 withdrawals. However, the ETH withdrawal finalization does not seem to be throttled. We raise this here for visibility since the finalization throttles are considered important by the team in terms of preventing massive withdrawals of a potential hack.		
P2	Throttling per user can be easily bypassed	**INFO**

The protocol modified Optimism's bridge adding throttling functionality over most of the bridging operations. The protocol defines the following throttle types:

- Per-user throttling, which limits the transfer that a user can make in a specified time interval.
- Global throttling, which uses a special address that is used for all the transfers regardless of who initiated the transaction.
- Global throttling, which is based on the `maxAmountTotal` that caps the transfers that a specific throttle can serve in total. This mechanism is dependent on the total balance of assets the contracts hold instead of relying on timely intervals.

However, the per-user throttling doesn't seem to be effective as it can be easily bypassed by anyone by splitting the funds into multiple accounts which can be used to bridge the assets all at once bypassing the time-based per-user throttling. Their operations will only be limited by the global throttle parameters.

P3	Users with excessively large balances (whales) can DoS the bridge	**INFO**

One of the purposes of the throttling mechanisms that were introduced with this update is to prevent bridging large amounts that may originate from hacks. The protocol utilizes 3 mechanisms which can be found on P2 above. However, the current implementation does not protect legitimate users from those who hold excessive amounts of funds (the so-called whales). These last users can continuously transfer large amounts through the bridge hitting the global throttle thresholds. Even though they cannot affect the per-user throttling they can move funds that reach the global limits effectively blocking others from using the bridge.

Regardless of the likelihood of such an incident, as the abusing users will have to lock their funds in the protocol for several hours, we mention this scenario here for visibility and possible mitigations in case required.

P4	Inconsistencies of the throttling mechanisms between the two directions of the bridge	**INFO**

The bridge defines throttling mechanisms for both initiating a bridge operation of funds and finalizing the bridging on the other side of the bridge. However, the two directions are not equivalent in terms of how the throttle mechanisms are used.

More precisely:

- In `L1StandardBridge`, the `_throttleETHInitiate` and `_throttleERC20Initiate` functions are called by the `bridge*` functions of `StandardBridge` to bridge the funds to `L2`.
    - On `L1`, the ETH deposits (aka. `L1 => L2`) are throttled on a per-user basis inside `L1StandardBridge` while they are also throttled globally inside `OptimismPortal` using the global user mechanism.
    - On `L1`, the ERC20 deposits (aka. `L1 => L2`) are throttled on a per-user basis inside `L1StandardBridge` and also globally, at the same time, by the `maxAmountTotal` mechanism.
    - On `L1`, the ERC20 finalizations (aka. `L2 => L1`) are throttled globally using the global user mechanism inside `L1StandardBridge`.
- In `L2StandardBridge`, the `_throttleETHInitiate` and `_throttleERC20Initiate` functions are called by the `bridge*` functions of `StandardBridge` to bridge the funds to `L1`. However, the applied throttles do

not follow the same mechanisms on `L2` compared to `L1`. For context, the previously called deposits, which bridged funds from `L1 => L2`, on `L2` are called withdrawals but they perform the same operation of bridging funds from the calling chain to a target chain (i.e. `L2 => L1` in this case). For example:

- On `L2`, the ETH withdrawals (aka. `L2 => L1`) are **not throttled at all** in the context of `L2StandardBridge` **when they were throttled on a per-user basis on L1**. They are, however, throttled globally inside `L2ToL1MessagePasser` using the global user mechanism.

- On `L2`, the ERC20 withdrawals (aka. `L2 => L1`) are **not** throttled on a per-user basis, but they are now **throttled globally inside `L2StandardBridge`** using the global user mechanism.

- On `L2`, the ERC20 finalizations (aka. `L1 => L2`) are **not throttled at all** inside `L2StandardBridge`. On the contrary, on `L1` the corresponding ERC20 finalizations are throttled globally using the global user mechanism.

- The above inconsistencies have also introduced discrepancies in the setter functions which update the throttle parameters. For example, `L2StandardBridge::setErc20ThrottleWithdrawalsMaxAmount` requires `maxAmountTotal` to be `0`, as it only uses the global user mechanism, but the corresponding `L1StandardBridge::setErc20ThrottleDepositsMaxAmount` does not apply such a restriction due to it using a per-user throttle mechanism. Similar observations apply to the other setter functions as well.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

[No low severity issues]

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Alternative way to prevent arbitrary initialization of the implementation contracts	**INFO**
All the contracts in scope are meant to be upgradeable. For that reason, they use initializers to set the state of the proxies after deployment, but they also call the initializers in the constructors of the implementation contracts to prevent their initialization by any random caller. However, you could call the		

`_disableInitializers` function inside the constructors instead of calling the initializers with default values.

A2	Redundant arguments are given to some throttle functions	**INFO**

In the `L1StandardBridge` contract, the `_throttleETHInitiate` and `_throttleERC20Finalize` functions call `_transferThrottling` on throttles which have not defined the `maxAmountTotal` value as a global throttle. As a result, the second argument of the `_transferThrottling` is redundant since it is only used by the `maxAmountTotal` global throttle if set. Thus, you can remove these arguments and always forward the value `0` instead which will also save some gas since the balance readings will be avoided.

`L1StandardBridge::_throttleETHInitiate:86`

```
function _throttleETHInitiate(
    address _from,
    uint256 _amount
) internal override {
    _transferThrottling(
      ethThrottleDeposits,
      _from,
       address(this).balance - _amount, // Dedaub: You can pass 0 instead
      _amount
    );
}
```

`L1StandardBridge::setEthThrottleDepositsMaxAmount:98`

```
function setEthThrottleDepositsMaxAmount(
    uint208 maxAmountPerPeriod,
    uint256 maxAmountTotal
) external {
    // we only perform per-user throttling of eth deposits since the
    // global cap is handled on the OptimismPortal
```

```
    require(maxAmountTotal == 0,
      "StandardBridge: max total amount not supported");
    _setThrottle(maxAmountPerPeriod, maxAmountTotal, ethThrottleDeposits);
    // Dedaub: maxAmountTotal is enforced to be 0 (aka. disabled)
}
```

Similarly, in `OptimismPortal`, the `depositTransaction` function uses the throttle with the global user configuration which makes `maxAmountTotal` redundant (see A3 for more). Thus, you could also forward the value `0` to the call of `_trasnferThrottling` similar to the call inside `finalizeWithdrawalTransaction` which also uses a throttle with the global user mechanism.

The same applies to `L2StandardBridge` for the `_throttleERC20Initiate` function which uses the global user as the throttle mechanism. However, for this case keep in mind the comments in P4 since it may require changes to its logic.

A3	Redundant assignment of `maxAmountTotal`	**INFO**

In `OptimismPortal`, only throttles for ETH deposits and withdrawals are used which use the global user as their mechanism. As a result, the `setEthThrottleDepositsMaxAmount` function does not require the `maxAmountTotal` to be `0`, compared to `setEthThrottleWithdrawalsMaxAmount` and the other setters in the `L*StandardBridge` contracts. `maxAmountTotal` does not make sense to be initialized in that context since only the global user throttling is applied and no per-user throttling exists.

A4	Possible checks could be added when configuring a throttle	**INFO**

In `TransferThrottle`, the `_setThrottle` function updates the parameters of a specific throttle. However, there are no checks to ensure that `maxAmountPerPeriod <= maxAmountTotal` since it wouldn't make sense to have `maxAmountPerPeriod >`

`maxAmountTotal` as the users may get more allowance, but the global cap will still restrict them from using their full availability which is capped by `maxAmountTotal`.

A5	Users could frontrun a throttle period length update	**INFO**

Users who have spent some of their available credits and notice a possible increment to the throttle's period length could try to bridge an infinitesimal amount just to realize their accumulated credits based on the previous period length preventing them from being affected by the period length change which would otherwise require more time to reach the same point of accumulated credits.

A6	Minor comment on the `StandardBridge` initializer	**INFO**

As a minor comment in the context of the current implementation, we mention here that the `onlyInitializer` modifier was removed from the `StandardBridge::__StandardBridge_init` function that is used to initialize the `StandardBridge` contract which both `L1StandardBridge` and `L2StandardBridge` inherit from. This does not cause any direct threats to the current state of the protocol since it is only reachable by the `initialize` functions which are nevertheless callable only once. We only mention this for future awareness in case a new version introduces calls to this function to keep in mind that they should always be restricted to avoid reinitialization of important variables of the protocol.

A7	Redundant initializer in `L2ToL1MessagePasser`	**INFO**

Inside `L2ToL1MessagePasser` an initalizer was introduced that is redundant as it sets the storage variable of both the proxy and the implementation to the same values. Either `_disableInitializers` should be called inside the initializer, or the initializer should be removed completely with the variable set as a constant.

`L2ToL1MessagePasser::initialize:69`

```
// Dedaub: Both proxy and implementation get the same values
```

```
constructor() { initialize(); }

function initialize() public initializer {
    accessController = AccessControlPausable(Predeploys.L2_CONTROLLER);
}
```

A8	Alternative way to allow only EOAs call specific functions	INFO

In the `StandardBridge` contract, the `onlyEOA` modifier is used on ETH bridge initiations to prevent accidental deposits from smart contracts. Currently, it only checks whether the calling address has code or not, but this does not protect against contracts calling from their constructors. It may not be an issue, as also stated in the comments in the code, but you could also check whether `tx.origin == msg.sender` which ensures that only EOAs can call the guarded functions.

A9	Inconsistent assumption in `L2OutputOracle`	INFO

In `L2OutputOracle`, the `setFinalizationPeriodSeconds` function allows the owner to change the `finalizationPeriodSeconds`. Moreover, in the comments it is stated that the value will never be desired to be larger than a week.

`L2OutputOracle::setFinalizationPeriodSeconds:167`

```
function setFinalizationPeriodSeconds(
    uint256 newFinalizationPeriodLength
) external onlySystemOwner {
    // We would never want a value that is larger than a week
    // (like for an OR) and want to avoid anyone being able to DoS the
    // bridge therefore we set an upper bound for the period length
    require(newFinalizationPeriodLength <= 31536000, // Dedaub: 1 year
      "L2OutputOracle: Finalization period too long");
    finalizationPeriodSeconds = newFinalizationPeriodLength;
}
```

However, the `require` statement uses the value of `31536000` which represents `1 year` in seconds instead of `1 week` or something closer to the assumption.

A10	Unclear change in Optimism's code	INFO

In the `OptimismPortal` contract of the official codebase of Optimism, the `finalizeWithdrawalTransaction` function had a `require` statement that checked if the output proposal had been finalized. However, the current changes seem to have removed that check with no particular reason to do so.

`@optimism/src/L1/OptimismPortal::finalizeWithdrawalTransaction:306`

```
// Check that the output proposal has also been finalized.
require(
    _isFinalizationPeriodElapsed(proposal.timestamp),
    "OptimismPortal: output proposal finalization period has not elapsed"
);
```

A11	Inconsistency in an access control configuration	INFO

In the `OptimismPortal`, the account to use for the access control and for the pausing of the protocol is of type `SuperchainConfig`. However, all other contracts in scope have changed this to be of type `AccessControlPausable`. Since `SuperchainConfig` is an enhanced `AccessControlPausable` we only mention this for the inconsistency among the contracts with no issues apparently.

A12	Compiler bugs	INFO

The code is compiled with Solidity `0.8.20`. Version `0.8.20`, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.